EE379V: Assignment #2

Due: Mar. 7 2022, 11:59pm TX

Introduction

The overall goal of this homework is to implement a local feature matching algorithm. You can discuss in small groups, but turn in individual solutions and indicate collaborators. Turn in assignments by Monday, Mar. 7. Submit to Canvas a .zip file containing (1) a questions.pdf with answers for Part 1 questions, (2) a writeup.pdf for Part 2 code; and (3) a `code/` directory containing your code without any large result files or subfolders. The pdf can be created using latex or converting a word document to pdf or any other method you prefer, as long as it is organized and easy to read. The submission template with latex and code for this homework can be found here: https://classroom.github.com/a/NbXwx05F.

Info: `Szeliski` refers to the second edition of *Computer Vision: Algorithms and Applications*, which can be found here.

1 Part 1 Questions (30%)

Instructions

- Three graded questions.
- Write code where appropriate.
- Feel free to include images or equations.
- **We do NOT expect you to fill up each page with your answer.** Some answers will only be a few sentences long, and that is okay.

1.1 Q1

The Harris Corner Detector is commonly used in computer vision algorithms to find interest points from which to extract stable features for image matching.

How do the eigenvalues of the 'M' matrix change if we apply a low-pass filter to the image? What if we apply a high-pass filter? Describe qualitatively.

1.2 Q2

Given an interest point location, the SIFT algorithm converts a 16×16 patch around the interest point into a 128×1 feature descriptor of the gradient magnitudes and orientations therein. Write pseudocode *with matrix/array indices* for these steps.

Notes Do this for just one interest point at one scale; ignore the overall interest point orientation; ignore the Gaussian weighting; ignore all normalization post-processing; ignore image boundaries; ignore sub-pixel interpolation and just pick an arbitrary center within the 16×16 for your feature descriptor. Please just explain in pseudocode how to go from the 16×16 patch to the 128×1 vector.

```
# You can assume access to the image, x and y gradients, and their magnitudes/
                                                    orientations.
image = imread('example.jpg')
grad_x = filter(image, 'sobelX')
grad_y = filter(image, 'sobelY')
grad_mag = sqrt( grad_x .^2 + grad_y.^2 )
grad_ori = atan2( grad_y, grad_x )

# Takes in a interest point x,y location and returns a feature descriptor
def SIFTdescriptor(x, y)
    descriptor = zeros(128,1)


    return descriptor
```

1.3 Q3

Distance metrics for feature matching.

(a) Explain the differences between the geometric interpretations of the Euclidean distance and the cosine similarity metrics. What does this tell us about when each should be used?

(b) Given a distance metric, what is a good method for feature descriptor matching and why?

2 Part 2 Code (70%)

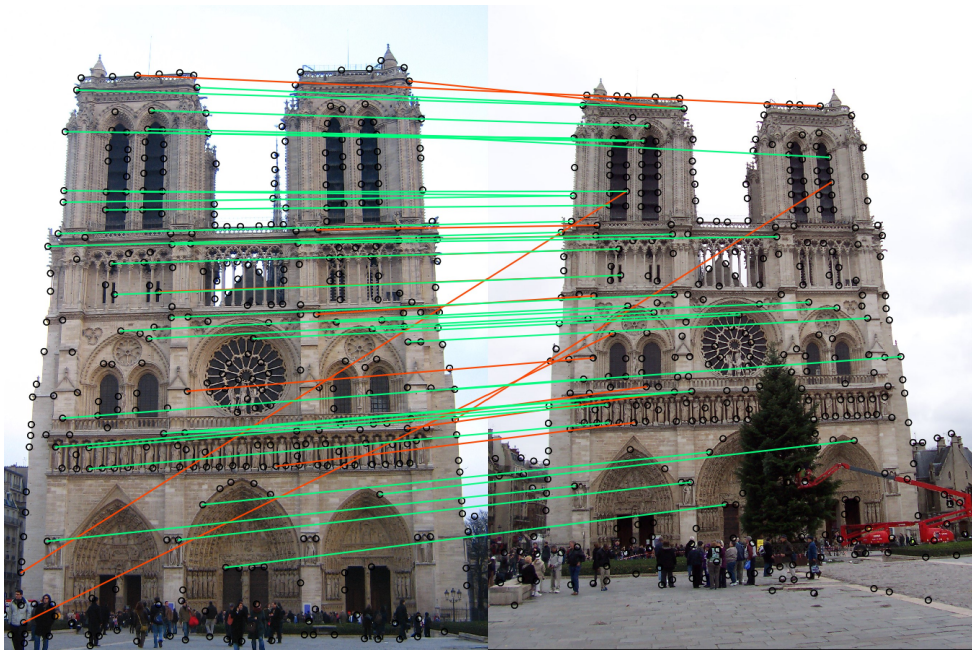


Figure 1: The top 50 most confident local feature matches from a baseline implementation of homework 2. In this case, 36 were correct (green lines) and 14 were incorrect (red lines).

Overview

We will create a local feature matching algorithm and attempt to match multiple views of real-world scenes. There are hundreds of papers in the computer vision literature addressing each stage. We will implement a simplified version of SIFT; however, you are encouraged to experiment with more sophisticated algorithms for extra credit!

Task: Implement the three major steps of local feature matching:

1. **Detection** in the `get_interest_points` function in `student.py`. Please implement the Harris corner detector (lecture 8 slides page 32). You do not need to worry about scale space invariance or keypoint orientation estimation for your Harris corner detector. You will also be implementing the `plot_interest_points` method to visually see where the interest points are located.

2. **Description** in the `get_features` function, also in `student.py`. Please implement a *SIFT-like* local feature descriptor (Szeliski 7.1.2). *You do not need to implement full SIFT!* Add complexity until you meet the rubric. To quickly test and debug your matching pipeline, start with normalized patches as your features.

3. **Matching** in the `match_features` function of `student.py`. Please implement the "ratio test" or "nearest neighbor distance ratio test" method of matching local features (Szeliski 7.1.3; equation 7.18 in particular).

Potentially useful functions: Any existing filter function, `zip()`, `skimage.measure.regionprops()`, `skimage.feature.peak_local_max()`, `numpy.arctan2()`.

Potentially useful libraries: `skimage.filters.x()` or `scipy.ndimage.filters.x()`, which provide many pre-written image filtering functions. `np.gradient()`, which provides a more sophisticated estimate of derivatives. In general, anything which we've implemented ourselves in a previous homework is fair game to use as an existing function.

Forbidden functions: `skimage.feature.daisy()`, `skimage.feature.ORB()`, and any other functions that extract features for you, `skimage.feature.corner_harris()` and any other functions that detect corners for you, any function which *computes histograms*, including `np.histogram()` and `np.digitize()`, `sklearn.neighbors.NearestNeighbors()` and any other functions that compute nearest neighbor ratios for you. `scipy.spatial.distance.cdist()` and any other functions that compute the distance between arrays of vectors (use the guide we've provided to implement your own distance function!). If you are unsure about a function, please ask.

2.1 Running the Code

`main.py` takes a command-line argument using `'-d'` to load a dataset, e.g., `'-d notre_dame'`. For example, `$ python main.py -d notre_dame`. Please see main.py for more instructions.

To define these arguments in Visual Studio Code for debugging, we need to create a Launch Configuration. More information on how to do that is in the VS Code documentation. In `launch.json`, we would add a configuration with the relevant input arguments `args` like this:

```
{
  "configurations": [
      ...,
      {
          "name": "Python: EE379V Homework 2",
          "type": "python",
          "request": "launch",
          "cwd": "./HW2_FeatureMatching/code",
          "program": "main.py",
          "args": ["-d", "notre_dame"],
          "console": "integratedTerminal"
      }
    ]
}
```

NOTE: Depending on which directory you open in VSCode, you may have to change your current working directory `cwd` variable. Using an absolute path is the most reliable method.

2.2 Requirements / Rubric

If your implementation reaches 60% accuracy on the *most confident* 50 correspondences in 'matches' for the Notre Dame pair, and at least 60% accuracy on the *most confident* 50 correspondences in 'matches' for the Mt. Rushmore pair, you will receive 70 pts (full code credit). We will evaluate your code on the image

pairs at **scale_factor=0.5** (`main.py`), so please be aware if you change this value. The evaluation function we will use is `evaluate_correspondence()` in `helpers.py` (our copy, not yours!). We have included this function in the starter code for you so you can measure your own accuracy.

Time limit: We will stop executing your code after 20 minutes. This is your time limit, after which you will receive a maximum of 40 points for the implementation. You must write efficient code—think before you write.

1. *Hint 1: Use "steerable" (oriented) filters to build the features.*
2. *Hint 2: Use matrix multiplication for feature matching.*

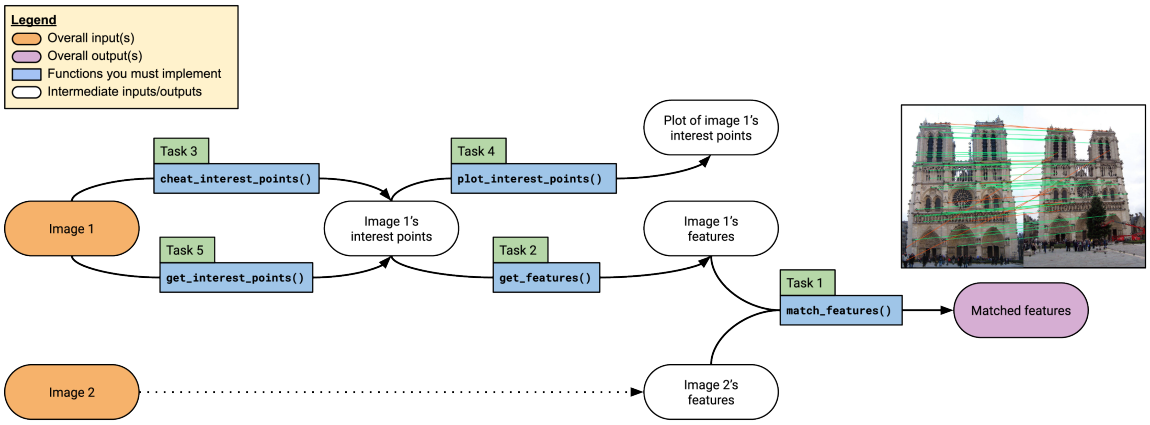
Compute/memory limit: We run on an Azure Standard D2s v3 with a modern desktop CPU about as powerful as your laptop. It does not have a GPU. Your job will have 768MB memory, and going over this may terminate your program unexpectedly (!). It has Ubuntu 18.04 and python 3.6.9 installed. If you are concerned about the memory usage of your program, you can check it by running python memusecheck in your code directory.

- +25 pts: Implementation of Harris corner detector in `get_interest_points()`
- +30 pts: Implementation of SIFT-like local feature descriptor in `get_features()`
- +10 pts: Implementation of "Ratio Test" matching in `match_features()`
- +05 pts: Writeup.
  - Please include your result images on the three scenes, describe any extra credit (please include the performance improvement), and tell us any other information you feel is relevant (e.g., any issues, problems, or implementation decisions that you'd like to highlight). The write up is not intended to be a detailed report; rather, a way for us to collect the required information for grading.
  - Show how well your method works on the Notre Dame, Mount Rushmore, and Episcopal Gaudi image pairs—the visualization code will write image-pair-specific output to your working directory, e.g., `notre_dame_matches.jpg`.
  - Provide plots of your interest points for each image on each of their respective images. Explain why your interest points may or may not look correct, and what surprises you (if anything).
- +10 pts: Extra credit (max 10 pts total).
  - Detection:
    - Up to 5 pts: Detect interest points at multiple scales or use a scale-selection method to pick the best scale.
    - Up to 5 pts: Use adaptive non-maximum suppression as discussed in the textbook.
    - Up to 10 pts: Use a different interest point detection strategy like MSER. Use it alone, or take the union of multiple methods.
  - Description:
    - Up to 3 pts: Experiment with SIFT parameters: window size, number of local cells, orientation bins, different normalization schemes.
    - Up to 5 pts: Estimate feature orientation.
    - Up to 5 pts: Multi-scale descriptor. If you are detecting interest points at multiple scales, you should build the features at the corresponding scales, too.
    - Up to 5 pts: Different spatial layouts for your feature (e.g., GLOH).
    - Up to 10 pts: Entirely different features (e.g., local self-similarity).
    - Up to 10 pts: Add KAZE features. KAZE features is a feature detection and description method similar to SIFT. However, where SIFT finds features in Gaussian scale, KAZE uses nonlinear scale space.
  - Matching; the baseline matching algorithm is computationally expensive, and will likely be the slowest part of your code. Consider approximating or accelerating feature matching:

    - Up to 5 pts: Create a lower dimensional descriptor that is still sufficiently accurate. For example, if the feature is 32 dimensions instead of 128 then the distance computation should be about 4 times faster. PCA would be a good way to create a low dimensional descriptor. You would need to compute the PCA basis on a sample of your local descriptors from many images.
    - Up to 5 pts: Use a space partitioning data structure like a kd-tree or some third party approximate nearest neighbor package to accelerate matching.

- -05*n pts: Where n is the number of times that you do not follow the instructions.

2.3 An Implementation Strategy



1. Implement `match_features()` . Accuracy should still be near zero because the features are random. To do this, you'll need to calculate the distance between all pairs of features across the two images (much like `scipy.spatial.distance.cdist()`). While this could be written using a series of `for` loops, we're asking you to write a matrix implementation using `numpy`. Writing things in `numpy` makes them orders of magnitude faster, and knowing how to do this is an important skill in computer vision. To help you with this implementation, we're providing a guide in the `match_features.pdf`. Make sure to also return proper confidence values for each match. Your most confident matches should have the highest values, and doing this wrong could cause the evaluation function to look at the wrong matches when grading your code.

2. Change `get_features()` to cut out image patches. Image patches are not invariant to brightness change, contrast change, or small spatial shifts, but this provides a baseline. Image patches are not ideal, but by using this preliminary implementation of `get_features()` and using `cheat_interest_points()` to get the relevant interest points, you can visualize the feature matches. You should be seeing accuracies of around 60-70% with this initial version of `get_features()` for the Notre Dame image pair (cheat interest points for the other two image pairs are not good). Then, finish `get_features()` by implementing a SIFT-like feature.

3. Use `cheat_interest_points()` instead of `get_interest_points()` to test whether your implementations of `get_features()` and `match_features()` work . At this point, just work with the Notre Dame image pair (the cheat points for Mt. Rushmore aren't very good and the Episcopal Palace is difficult to feature match). Accuracy should increase to 70% on the Notre Dame pair. This function cannot be used in your final implementation. It directly loads the 100 to 150 ground truth correspondences for the test cases. Even with this cheating, your accuracy will initially be near zero because the features are random and the matches are random.

4. Finish `plot_interest_points()` to see what the cheat interest points look like for the Notre Dame image pair. Keep note of what these interest points look like and compare them to what you see when you implement `get_interest_points()`.

5. Stop cheating (!) and implement `get_interest_points()`. Harris corners aren't as good as ground-truth points (which we know to correspond), so accuracy may drop. On the other hand, you can get hundreds or even a few thousand interest points, so you have more opportunities to find confident matches. Our solution generates around 300–700 interest points for each image.

These interest point and accuracy numbers are only a guide; don't worry if your method doesn't exactly produce these numbers. Feel confident if they are approximately similar, and move on to the next part.

2.4 Notes

`main.py` handles files, visualization, and evaluation, and calls placeholders of the three functions to implement. For the most part you will only be working in `student.py`. Please don't change the folder structure by renaming or moving the files.

The Notre Dame, Mount Rushmore, and Episcopal Gaudi image pairs include 'ground truth' evaluation. `evaluate_correspondence()` will classify each match as correct or incorrect based on hand-provided matches. You can test on those images by changing the `-d` argument you pass to `main.py`.

As you implement your feature matching pipeline, check whether your performance increases using `evaluate_correspondence()`. Take care not to tweak parameters specifically for the initial Notre Dame image pair. We provide additional image pairs in extra_data.zip (194 MB), which exhibit more viewpoint, scale, and illumination variation. With careful consideration of the qualities of the images on display, it is possible to match these, but it is more difficult.

You will likely need to do extra credit to get high accuracy on Episcopal Gaudi.

3 FAQ

Q: I have implemented my get_interest_points() function, but it seems the evaluation function shows very low accuracy on both TOP-50 and TOP-100 matches.
A: This is usually because there might be a confusion of axis x and y. The returned values xs and ys specify coordinates in the image, so ys should correspond to rows and xs correspond to columns. It means we might want to flip the order of the two returned values.
Q: I have implemented my get_interest_points() function, but it seems the evaluation function shows 0% accuracy on TOP-100 matches or both TOP-50/100 matches.
A: It might be caused by not returning enough matches. We need to make sure that we are returning at least 100 matches in any case. The evaluation function tries to compare TOP 100 matches for testing accuracy.
Q: I found that there are negative Harris cornerness scores in my results.
A: No worries, cornerness scores could be negative.

4 Linear Algebra Review

For this homework, it may be beneficial to review eigenvalues and eigenvectors.